Int J Parallel Prog (2016) 44:208–232
DOI 10.1007/s10766-014-0312-y

Architectural Support for Fault Tolerance in a Teradevice Dataflow System

Sebastian Weis · Arne Garbade · Bernhard Fechner · Avi Mendelson · Roberto Giorgi · Theo Ungerer

Received: 28 February 2013 / Accepted: 7 May 2014 / Published online: 29 May 2014

Abstract The high parallelism of future Teradevices, which are going to contain more than 1,000 complex cores on a single die, requests new execution paradigms. Coarse-grained dataflow execution models are able to exploit such parallelism, since they combine side-effect free execution and reduced synchronization overhead. However, the terascale transistor integration of such future chips make them orders of magnitude more vulnerable to voltage fluctuation, radiation, and process variations. This means dynamic fault-tolerance mechanisms have to be an essential part of such future system. In this paper, we present a fault tolerant architecture for a coarse-grained dataflow system, leveraging the inherent features of the dataflow execution model. In detail, we provide methods to dynamically detect and manage permanent, intermittent, and transient faults during runtime. Furthermore, we exploit the dataflow execution model for a thread-level recovery scheme. Our results showed that redundant execution of

S. Weis (✉) · A. Garbade · B. Fechner · T. Ungerer
University of Augsburg, Universitaetsstr. 6a, 86159 Augsburg, Germany
e-mail: sebastian.weis@informatik.uni-augsburg.de

A. Garbade
e-mail: garbade@informatik.uni-augsburg.de

B. Fechner
e-mail: fechner@informatik.uni-augsburg.de

T. Ungerer
e-mail: ungerer@informatik.uni-augsburg.de

A. Mendelson
Technion, Technion City, 32000 Haifa, Israel
e-mail: avi.mendelson@technion.ac.il

R. Giorgi
University of Siena, Via Roma 56, 53100 Siena, Italy
e-mail: giorgi@dii.unisi.it

dataflow threads can efficiently make use of underutilized resources in a multi-core, while the overhead in a fully utilized system stays reasonable. Moreover, thread-level recovery suffered from moderate overhead, even in the case of high fault rates.

Keywords Coarse-grained dataflow · Fault tolerance · Fault detection · Recovery · Reliability

1 Introduction

Nowadays, the number of transistors per chip still increases and current devices such as Nvidia's Fermi architecture already incorporate over 3 Billion transistors [47]. However, improvements in performance are no longer driven by significant enhancements of the clock rates or the exploitation of instruction-level parallelism. Instead, scalable chip multiprocessor systems (CMPs) are gaining ground [7,19,23]. Technology forecasts predict that future parallel computing systems may contain more than 1,000 complex cores (Teradevices) per die [10] and request new execution paradigms to efficiently exploit their large amount of parallelism.

Dataflow execution models are known to overcome the limitations of the traditional von Neumann architecture by leveraging the inherent parallelism of the applications. However, early word-based dataflow architectures suffered from high synchronization overhead and low per-instruction performance [26]. Therefore, coarse-grained dataflow models were developed to reduce the synchronization overhead and exploit the efficient sequential execution of current processors, while still providing enough parallelism and efficient thread-level synchronization mechanisms. In particular, with the rise of many-core processors, coarse-grained dataflow architectures and compilers gained new attention in academia [12,15,17,20,27,44,49].

On the other side, the International Technology Roadmap for Semiconductors [1] prognoses that the shrinking feature size of future chips and decreasing supply voltage leads to increasing failure rates [43]. Also, complexity and costs for testing and verification of devices will increase. With the ongoing decrease of the transistor size, the probability of physical flaws on the chip, induced by voltage fluctuation, cosmic rays, thermal changes, or variability in the manufacturing process will further raise [43], making faults in present multi-core and future many-core systems unavoidable. While fault tolerance has always been essential in safety–critical systems [48], the architecture of a general purpose processor is much stronger influenced by economical constraints. Therefore, future many-core systems will require fault-tolerance techniques, which are capable to scale with the number of cores and the increasing failure probability on a chip in conjunction with a reasonable architectural effort [9].

Consequently, we argue that coarse-grained dataflow is not only a promising candidate to exploit parallelism in future Teradevices, but can also serve as a basis for an efficient fault-resilient parallel architecture. The pure single-assignment and side-effect free semantics of dataflow threads provide an advantage for efficient recovery and redundant execution mechanisms compared to state-of-the-art multi-core systems.

In this paper, we will present a parallel coarse-grained dataflow system, which is enhanced with redundant execution and thread-level recovery mechanisms.

The presented architecture will be able to cope with the following fault types:

- Detect and manage permanent or intermittent faults in on-chip devices by health monitoring.
- Manage and react on core-internal faults in components safeguarded by the Machine Check Architecture.
- Detect and recover form permanent, intermittent, and transient faults within the cores by a dataflow-based redundant execution and recovery approach.

In the following, we will call the redundant execution of coarse-grained dataflow threads *double execution*.

The additional contributions of this paper compared to [46] are:

1. We describe the architectural implications of redundant execution and thread-level recovery for the coarse-grained dataflow runtime system.
2. We evaluated and quantified the overhead of the redundant execution scheme and thread-level recovery for two benchmarks.
3. We quantified the overhead of thread-level recovery in a system, suffering from faults.

The remainder of the paper is organized as follows: in Sect. 2 we present prior work related to fault tolerant dataflow architectures, redundant execution, and recovery. Section 3 describes the underlying dataflow architecture. Based on this, we describe the architectural extensions required for fault management, fault detection, and recovery in Sect. 4. In Sect. 5, we describe the double execution mechanism and present implications for the architecture. The mechanism to recover from faults is presented in Sect. 6. In Sect. 7, the overhead for double execution and thread-level recovery in the case of faults is quantified, followed by a conclusion in Sect. 8.

2 Related Work

This section presents prior work on fault detection and recovery on architectural level as well as on coarse-grained dataflow architectures.

2.1 Fault Tolerance in Coarse-Grained Dataflow Architectures

The benefits of a side-effect free execution model for fault tolerance have already been studied in the context of different dataflow architectures. Nguyen et al. [29] proposed a fault tolerance scheme for a wide-area parallel system, considering a macro dataflow architecture built on top of a wide-area distributed system. This differs from our architecture as we target a single chip multiprocessor system with hardware support for thread scheduling and fault tolerance.

Another technique by Jafar et al. [22] exploits the macro dataflow execution model of KAAPI [13] for a checkpoint/recovery model. KAAPI uses a C++ library on commodity chip multiprocessor clusters that exposes a dataflow programming model.

Since KAAPI is a software library, the model has to cope with the overhead usually introduced with software fault tolerance techniques. Our work mainly focuses on hardware fault tolerance mechanisms.

2.2 Redundant Execution

Tightly-coupled lockstepping systems [39,48] are well-known redundant execution mechanisms and efficiently used in safety-critical systems and transaction processing systems. Nevertheless, tightly-coupled lockstepping requires synchronization on instruction or memory access granularity. This fine-grained synchronization requests a highly deterministic execution and therefore complicates the use of tightly-coupled lockstepping for parallel applications and impedes the use of aggressive out-of-order cores as well as power saving mechanisms like dynamic voltage and frequency scaling (DVFS) [8]. On microarchitectural level different mechanisms were proposed to exploit underutilized resources in superscalar processors for efficient redundant execution. Austin [6] proposed DIVA, an additional pipeline stage to verify the execute stage of an out-of-order processor. Ray et al. [32] proposed a scheme that dynamically replicates instructions in an out-of-order pipeline.

Rotenberg [34] was the first, who used simultaneous multithreaded processors for redundant execution on thread-level. In [33], Reinhardt et al. extended Rotenberg's approach by widening the fault coverage and presenting performance improvements. In [28], Mukherjee et al. studied redundant execution for a commercial-grade simultaneous multithreaded processor. They proposed a redundant execution scheme for a dual-core processor, which is able to detect both transient and permanent faults.

All previously mentioned thread-level redundant techniques provide fault detection for sequential applications. However, redundant execution of parallel applications poses additional difficulties, since critical sections and atomic operations complicate a consistent memory view of redundantly executed threads [31,35,38].

With the ongoing technology scaling, different thread-level redundant execution schemes for multi-core processors were proposed, which does not require the fine-grained synchronization of tightly-coupled lockstep system, while they additionally provide redundant execution of parallel applications [25,31,35,36,38,40].

Our double execution approach of coarse-grained dataflow threads (see Sect. 5) is a loosely-coupled thread-level redundant execution scheme, which exploits the dataflow execution model and provides support for parallel dataflow applications in order to detect permanent, intermittent, and transient faults.

2.3 Rollback-Recovery Mechanisms

Elonzahy et al. [11] divide rollback-recovery for message-passing systems into *checkpoint-based* and *log-based* mechanisms. Checkpointing depends on restoring a global system state, while log-based mechanisms combine checkpointing with logging of non-deterministic events.

Prvulovic et al. [30] and Sorin et al. [42] both described global checkpointing techniques with logging for the rollback-recovery in a shared memory multiprocessor.

Recently, Agarwal et al. [2] presented a local checkpointing scheme for highly parallel systems using directory-based coherence.

Since our execution model provides inherent checkpoints between dataflow threads (see Sect. 3.2), we use a local thread-restart mechanism without the need for restoring a global state or the logging of events or messages.

3 Overall Architecture

Our assumed dataflow architecture and execution model builds upon the Decoupled Threaded Architecture (version for clustered architectures DTA-C) originally described in [15]. DTA-C was designed to fully exploit the Thread-Level Parallelism (TLP) provided by future parallel systems. Although, it is based on DTA-C, our architecture differs in two points from the DTA-C approach. First, unlike in DTA-C, in this paper we do not imply a special purpose synchronization and execution pipeline, but a standard x86–64 pipeline per core. Second, the coarse-grained dataflow threads are composed of standard x86–64 instructions, including also control flow instructions to support micro control flow within a thread. This allows the dataflow execution model to exploit data locality for the sequential execution of threads without replicating instructions or having to create new threads. For simplicity, we call a coarse-grained dataflow thread with micro control flow support a *thread*.

3.1 Basic Architecture

We assume a tiled hardware architecture, where a tile is denoted as a node. As shown in Fig. 1 each node is comprised of a certain number of cores and node management modules. In the following, we will describe these components in detail.

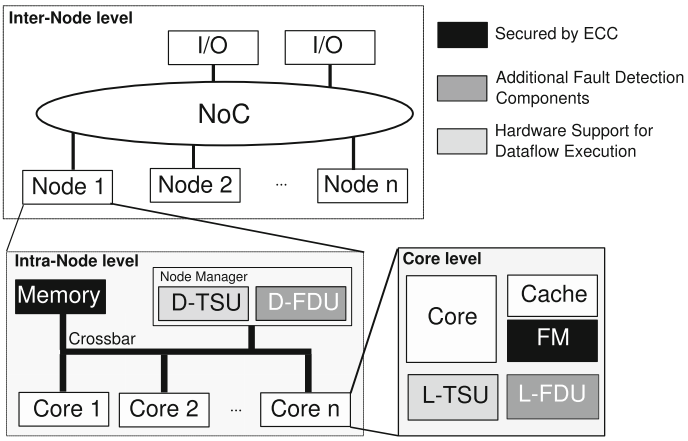

Fig. 1 High-level architecture

3.1.1 Core Level

On the core level, the basic elements of the architecture are single cores containing an x86–64 pipeline (x86–64 ISA with dataflow extensions derived from [14]) along with a private two-level cache hierarchy. Each core includes special hardware extensions consisting of two modules:

- The *Local Thread Scheduling Unit* (L-TSU) is responsible for scheduling threads on its affiliated core and communicating with the node's D-TSU.
- The *Local Fault Detection Unit* (L-FDU) is responsible for the detection of faults and reliability management within a core.

Beside the L-TSU and L-FDU, each core stores the input data of a running thread in the Frame Memory (FM). The Frame Memory is managed in a way that the data appears at the top level of the memory hierarchy (possibly all in the L1 cache [16]). This memory is filled with the thread's data (denoted as *thread frame*) before execution. Threads are not allowed to read from other thread frames. However, write operations into disjoint locations are permitted to support communication between threads in order to provide the inputs for subsequent threads.

3.1.2 Intra-node Level

From the intra-node perspective, we propose two additional hardware modules for management purposes. First, the *Distributed Thread Scheduling Unit* (D-TSU) coordinates the scheduling of the threads to cores within a node and communicates with other D-TSUs. Therefore, the D-TSU holds a table for bookkeeping the thread-to-core relations. Second, the *Distributed Fault Detection Unit* (D-FDU) is responsible for fault detection, performance monitoring, and reliability management within a node.

The additional hardware overhead for the dataflow thread scheduling support has been estimated in [4]. The authors conclude that the hardware overhead for dataflow thread scheduling in an 8 core node ranges from approx. 1–10 % compared to 512 kB of total on-chip cache memory.

3.1.3 Communication

For the communication between nodes, we assume a scalable 2D mesh-based interconnection network. All communication from one node to another will be handled by the interconnection network. Furthermore, we consider memory controllers to access off-chip DRAM and I/O-controllers on inter-node level. The controllers are connected to the interconnection network as well.

3.2 Execution Model

A dataflow program is partitioned in coarse-grained dataflow threads, where the execution of a thread consists of three phases. First, the pre-load phase, where the data is prefetched into the FM of the executing core. The second phase is the thread execution,

where the thread executes while only accessing its private heap and stack. The third phase is the post-store phase, where the results of the thread execution are distributed to the consuming thread frames.

Beside the frame, each thread has an assigned control structure called *continuation*. This structure stores control information about the thread, i.e. the pointer to the thread frame, the program counter, and the synchronization count (number of empty inputs). The continuations of a node are all managed within the D-TSU and the L-TSUs within a node. A thread will be scheduled for execution if and only if all inputs have been written to the thread's frame and therefore its synchronization count is zero.

Since in DTA-C prefetching can be very productively coupled with the scheduling of threads, accesses to the FM usually have low latency and are not likely to suffer from page faults or cache misses. Generally, a core's pipeline is supposed to rarely stall in the case of FM accesses.

Although, the baseline architecture uses commodity x86–64 pipelines, it requires additional instructions to control the dataflow thread execution by the D-TSUs and L-TSUs and to allow write and read access to the threads' frame memories. We use a minimal subset of the T*-instruction set extension, described in [14]. These instructions are:

- TSCHEDULE instructs the D-TSU to schedule a new thread. After the thread has been scheduled, the D-TSU returns the frame pointer to the calling core.
- TDESTROY indicates the end of a dataflow thread. After reception, the D-TSU releases all resources acquired by this thread.
- TREAD allows a thread to read data from its own FM. Memory accesses to other thread frames are prohibited.
- TWRITE enables a thread to write to the FMs of subsequent threads and decreases the synchronization count in the threads' continuations.

4 Architectural Support for Fault Detection and Recovery

In this section, we will describe the hardware extension to support fault detection and recovery in the baseline architecture. The central components of our dynamic fault detection approach are the already mentioned Fault Detection Units (FDUs). The Distributed FDU (D-FDU) is a lean hardware unit operating as an observer–controller on intra-node level. As such, a D-FDU autonomously queries and gathers the health states of all cores within its node. In this context the D-FDU is supported by the L-FDUs (described in Sect. 4.2) located with each node's core. In addition, D-FDUs monitor each other in order to detect faults of other D-FDUs in other nodes. The D-FDU dynamically analyzes the gathered information and provides the thread scheduler on intra-node level (D-TSU) with information about the state of the whole node and other D-FDUs.

4.1 Faults in Future Teradevices

We assume that fault rates will rise in all components of a terascale device. Therefore, it must be assumed that in the future a many-core will more likely suffer from transient

permanent, or intermittent faults in cores, interconnects and on-chip memories than current processors.

Furthermore, faults of all types are presumed to occur in different components at a time. At this stage of development, a component can be a D-TSU, an L-TSU, a D-FDU, an L-FDU, a core, or a link.

More detailed, on intra-node level, we assume (1) permanent, intermittent, and transient faults within cores and L-FDUs and (2) permanent and intermittent broken links between cores, L-TSUs, and L-FDUs.

On inter-node level, our architecture has to cope with permanent, intermittent and transient faults of whole nodes or links between nodes, I/O, and memory.

To summarize, our architecture will be able to detect the following fault types:

1. Permanent or intermittent D-FDU, L-FDU, or link faults by health monitoring.
2. Core-internal faults in components safeguarded by the Machine Check Architecture (MCA, see Sect. 4.4).
3. Permanent, intermittent, and transient faults within a core by double execution.

Finally, it provides recovery from all fault types in the cores by re-execution of dataflow threads.

4.2 L-FDU

The L-FDU is a small hardware unit implemented on each core to detect transient faults by extracting information from the MCA. Basically, the L-FDU has two tasks: (1) Reading out the fault detection registers of the monitored core, i.e. registers of the Machine Check Architecture. (2) Periodic communication with the D-FDU by sending health messages from the core.

4.3 D-FDU

Concerning intra-node fault detection, the D-FDU detects core faults and informs the D-TSU about faulty components, while the D-TSU is responsible for double execution and thread restarts.

For the internal behavior of the D-FDU, we adopt an autonomic computing approach [24] organizing the operation principle into the four consecutive steps: Monitoring, Analyzing, Planning, and Executing (cf. Fig. 2). This MAPE cycle operates on a set of managed elements, comprising node (cores and D-TSU) and inter-node level elements (other D-FDUs) in other nodes [45].

D-FDUs detect faults and proactively maintain the operability of the node they monitor, for example by dynamically performing clock and voltage scaling while monitoring the cores' error rates, temperatures, and utilization. In this context, proactive means the prediction of a core's health state based on monitored information and taking action before the core gets damaged.

The intra-node monitoring of the cores, the D-TSU, and D-FDU is separated in two categories: time and event-driven. Time-driven messages are heartbeat messages that contain a set of core health information. Of particular interest are faults that influence

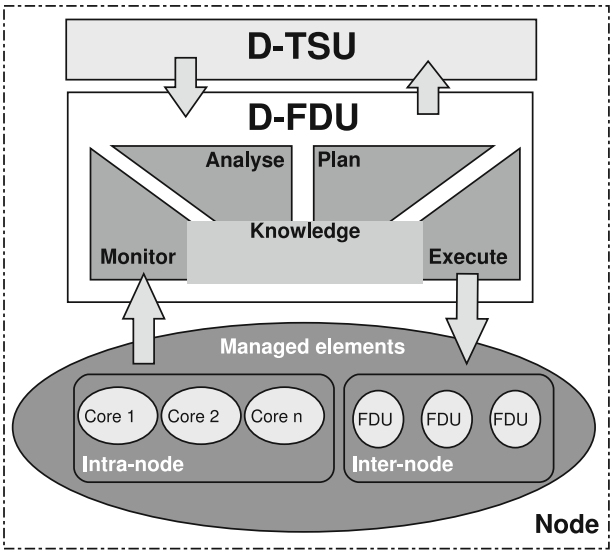

Fig. 2 MAPE cycle of the D-FDU

the actual core performance. The D-FDU expects a heartbeat message of a core in a certain time interval. If no heartbeat messages arrive at the D-FDU within the expected interval, the associated core will be suspected as faulty. When a permanent fault can be assured, e.g. multiple faults are detected in a short period of time; the D-FDU considers the core as completely broken. As a consequence, the D-FDU marks the core as unavailable and informs the D-TSU. The D-TSU itself is monitored by the D-FDU with the same techniques as a regular core. Thus, D-TSU faults can be detected as well.

The D-FDU communicates with the D-TSU via command messages, i.e. notify, request, and response messages. The D-TSU requests the D-FDU to change the frequency of a core or to reduce the frequency in the case of low workload, while the D-FDU reports the D-TSU on thermal and error conditions. In case of an intermittent or permanent error, the D-TSU temporarily or permanently stops scheduling any threads to the broken core.

D-FDUs can suffer from faults as well. To distribute reliability information between nodes, D-FDUs monitor each other.

Event-driven messages are alert messages sent by L-FDUs to their affiliated D-FDU. Such messages are sent when an urgent error report is needed in order to maintain the system stability. As an example, a core must be shut down due to critical thermal conditions, which may also influence neighboring cores.

4.4 Core-Internal Fault Detection

Most recent microprocessors are equipped with an architectural subsystem called Machine Check Architecture (MCA) [21], which is able to detect and correct certain

faults. For instance, for the AMD K10 processor family [3], the MCA can detect faults in the data and instruction cache, the bus unit, the load-store unit, the northbridge, and the reorder buffers. Since most space on current chips is occupied by large memory arrays, it is likely that in the case of error a memory cell will be affected, finally resulting in a machine check exception raised by a core's machine check architecture.

We assume that all cores include a minimal Machine Check Architecture, which checks and corrects faults in registers, Frame Memory, caches and the off-chip DRAM.

Since frequent occurrences of correctable and non-correctable faults may be a direct indicator for intermittent or permanent faults, or a permanent breakdown of the whole core, the L-FDUs transmit this information within their periodic heartbeat messages to the D-FDU. The D-FDU uses then the information to make predictions about the current reliability state of the core.

5 Double Execution

The dataflow execution model simplifies the duplication of threads during the execution. We follow the definitions given by Rotenberg [34] and call the thread that is duplicated *leading thread* and its copy *trailing thread*. Please note that we use this terminology only to distinguish between threads, those threads actually do not need to be executed subsequently.

Since the execution of dataflow threads is side-effect free and writes are only assigned once, we must only duplicate the *continuation* of a thread. This relaxes the complexity for the memory management as well as the management of the trailing thread.

Redundant execution of dataflow threads promises the following advantages over conventional lock-step architectures:

1. Result comparison can be restricted to data, which is consumed by subsequent threads.
2. Result propagation is only required when a thread has finished execution, which inherently supports deferred memory updates.
3. Redundant threads are synchronized at thread level. This enables the exploitation of the scalability of the dataflow model for redundant thread pairs. In particular, the D-TSU scheduler can take advantage of underutilized cores.

Figure 3 shows the dependency graph of a coarse-grained dataflow application (left) and the dynamically created dependency graph of the same application during a double execution run (right). It can be seen that the original program first executes T_0. Since this part of the program is sequential, double execution may exploit underutilized cores for spatial redundant execution of T_0', if the system has at least more than one core. After the results of T_0 and T_0' are compared, the synchronization counts of the subsequent threads are decremented, and the threads $(T_1 T_1', \ldots, T_n T_n')$ can be started. In this case, the hardware thread scheduler will try to execute as many redundant thread pairs (T_n and T_n') as possible in a spatial redundant way (see Fig. 4, left). If it is not possible to schedule all waiting threads to idle cores, since the thread-level parallelism of the original program was already able to utilize all cores, the scheduler will execute the threads in a temporal redundant and a spatial redundant way (see Fig. 4, right).

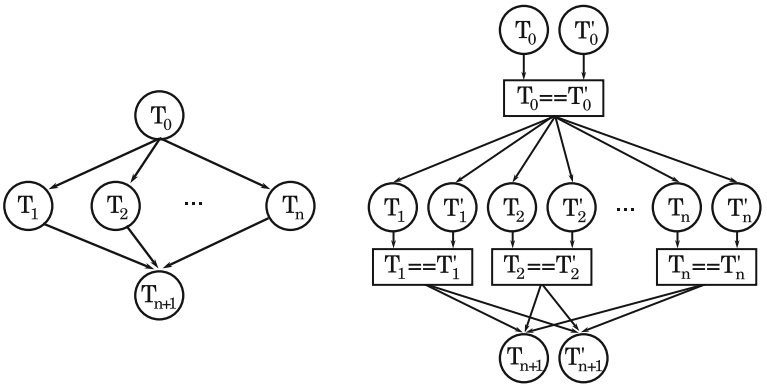

Fig. 3 Dependency graphs of regular dataflow execution (*left*) and double execution (*right*)

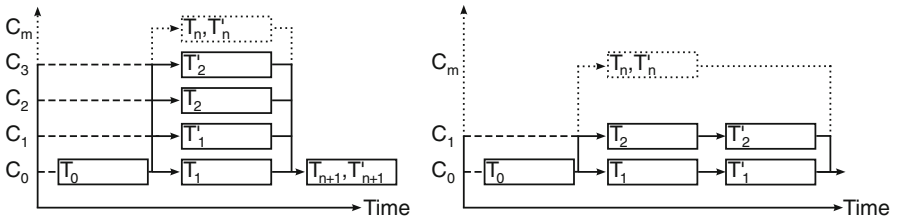

Fig. 4 Dynamic spatial and temporal redundancy of double execution

Within the Thread-to-Core List (TCL) in each D-TSU, all continuations scheduled to a core within the node are redundantly stored. Since the thread frame of dataflow thread is immutable after the synchronisation count has reached zero, the redundant threads are allowed to read the input data from the same thread frame, because data inconsistencies between redundant threads induced by race conditions of concurrent TREAD and TWRITE operations are impossible. Therefore, double execution only needs to duplicate the continuation of a thread. After the continuation has been copied, both threads can be independently scheduled to different cores for execution, while sharing the same thread frame as input data.

This means we can additionally exploit data locality by sharing the thread frame between the leading and the trailing thread.

The L-FDUs reduce the result set per thread to a 32-bit signature and forward it to the node's D-FDU, which compares all signature pairs. The D-FDU signals the D-TSU the commitment of the leading thread. In the fault free case, the results of the leading thread are forwarded by the L-TSU to the D-TSU and stored in all consuming thread frames.

Otherwise, the D-TSU has to trigger the recovery mechanism, described in Sect. 6. In more detail, double execution works as follows:

1. A thread is duplicated when its synchronization count becomes zero, i.e. a thread has received all its inputs and is ready to execute. To indicate the thread's duplication, the L-TSU sends notification messages to the D-TSU and the D-FDU. The

D-TSU, which keeps all continuations of its node in the TCL, creates a copy of the leading thread's continuation, distributes it to another core's L-TSU and stores the new continuation in the TCL. The thread distribution is limited to the cores affiliated to the D-TSU. However, a leading thread and a trailing thread never share the same core. This rule is enforced by sending the specific continuation to different cores within the node. The respective L-TSU proceeds with the execution of the leading thread as usual.

2. During execution, the L-TSU buffers all TWRITEs of the leading thread in a core-local write buffer (for more details see Sect. 5.1). Simultaneously, the L-FDU creates a CRC-32 signature of all TWRITEs, incorporating the target thread ID, the target address, and the data. The L-FDU of the trailing threads core also creates a CRC-32 signature of all TWRITEs; however, the TWRITEs of the trailing thread are discarded afterwards. Other write operations to the thread local storage (heap or stack) do not need to be buffered, since these writes will be automatically repeated by a thread restart.
3. When a thread has finished execution, indicated by a TDESTROY instruction, the core's L-FDU sends the CRC-32 signature to the D-FDU.
4. The D-FDU waits for the signatures of both the leading and the trailing thread, compares them, and informs the D-TSU about the result.
5. In the case of a non-faulty execution of both threads, the L-TSU redirects the buffered writes of the leading thread to the D-TSU, which commits them to the main memory and reduces the synchronization counts of the successor threads. Finally, the responsible D-TSU subsequently deletes the continuations of the leading and the trailing thread in its TCL. If a fault was detected, the D-TSU instructs the L-TSU to flush the core-local write buffer of the leading thread and discards all continuations created by the faulty thread.

5.1 Runtime Extensions for Double Execution Support

Intra-node thread management and scheduling are controlled by the D-TSU. Therefore, the D-TSU maintains all continuations of its node. Beside the synchronization count, each continuation keeps pointers to its code, input thread frame, and thread local storage. Because of the dataflow execution model, the execution of a thread is side-effect free. Therefore, from a D-TSU point of view, we only need to duplicate the thread's continuation in the D-TSU to create a trailing thread. The execution model guarantees that the code and the input thread frame will never change during thread execution. The thread local storage, however, is unique to each thread, and newly allocated with each continuation. The double execution mechanism takes advantage of the execution model, since this way only TWRITEs can release results of a thread to the global system state and hence need to be incorporated in the CRC-32 signature calculation. Figure 5 shows the original and the doubled continuation, with the additionally allocated thread local storage for the trailing thread. We added the following fields to the original continuation:

– RED_CONT (Redundant Continuation Pointer): Stores the thread ID of the redundant continuation.

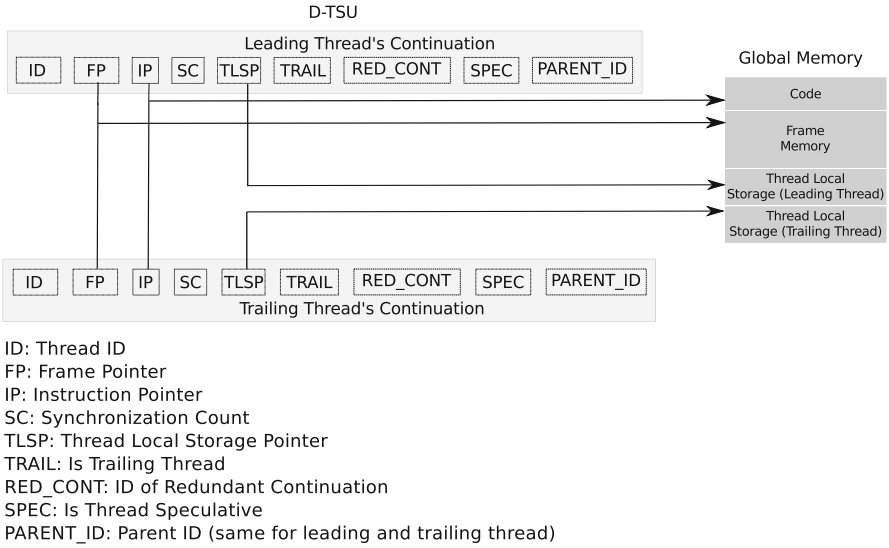

Fig. 5 Redundant continuations and memory organization

– SPEC (Speculative): Indicates whether this thread is speculative.
– PARENT_ID: Stores the thread ID of the parent thread. Required to discard the created speculative continuations, if the parent thread needs to be restarted.
– TRAIL: Indicates whether this thread is a trailing thread.

Please note, that the synchronization count of the redundant thread is always zero, since the redundant continuation is only created, when the synchronization count of the original thread became zero. As mentioned before, the trailing thread is handled by the D-TSU scheduler like a usual thread, with the exception that redundant threads will never be scheduled to the same core, to enable the detection of both permanent and transient faults. Furthermore, trailing threads are never moved to other nodes to ensure a fast result comparison by the node's D-FDU.

Although threads do not need to wait for input data after the synchronization count became zero, they can issue TSCHEDULE instructions to dynamically create subsequent dataflow threads. If the D-TSU receives a TSCHEDULE request from an L-TSU, a new continuation is created and a new frame memory, required to store the threads data, is allocated. The D-TSU finally returns the ID of the newly created thread. However, the returned ID is runtime dependent and may be later passed to subsequent threads by TWRITE instructions. To ensure deterministic write sets of the redundant threads, the returned IDs must be equal for both the trailing and the leading thread (see Fig. 6). Since the thread execution is not synchronized on instruction level, it may happen that the leading thread runs behind the trailing thread or vice versa. In order to prevent stalls induced by TSCHEDULE synchronization between redundant threads, we let both cores issue TSCHEDULE instructions. To prevent redundant thread creation, the D-FDU maintains a counter per continuation, which keeps the number of issued TSCHEDULE instructions. When a thread issues a TSCHEDULE request, the

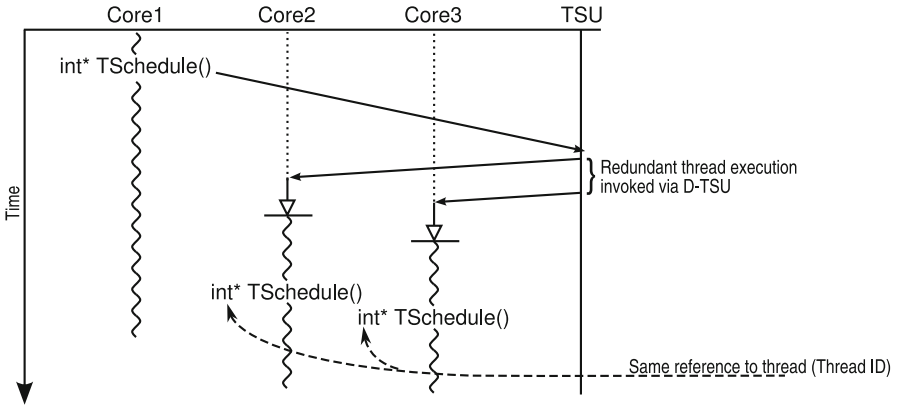

Fig. 6 Input replication during double execution for the TSCHEDULE instruction

D-TSU compares the TSCHEDULE count in its continuation with the TSCHEDULE count of the redundant continuation. When the TSCHEDULE count is greater than the TSCHEDULE count of the redundant thread, the D-TSU knows that this thread is running ahead of its redundant copy. In this case, the D-TSU processes the TSCHEDULE request as usual and stores the created thread ID in a table. If the TSCHEDULE count is lower than the TSCHEDULE count in the redundant continuation, the D-TSU has already processed this TSCHEDULE request from the redundant thread, which is running ahead. In this case, the D-TSU proceeds with a table lookup to retrieve the previously stored thread ID, which was created by the TSCHEDULE of the redundant thread. Furthermore, all continuations created by TSCHEDULE instructions are initially marked as speculative. This is necessary for the thread restart mechanism, because the D-TSU must ensure that all continuations created by a faulty thread can be discarded. In order to do so, the parent thread's ID is stored with each continuation. In the case of a thread restart, the D-TSU traverses all continuations and deletes continuations created by this thread. Furthermore, the D-TSU releases the allocated thread frames of these threads. Please note, that the D-TSU will not schedule a thread to a core, when its continuation is marked as speculative. If the D-FDU confirms the fault free execution of the thread, the D-TSU will mark all successor threads as non-speculative.

5.2 Core Level Enhancements

Although the dataflow execution model provides side-effect free execution, the underlying architecture must additionally guarantee fault isolation. From a pessimistic point of view, we must assume that TWRITEs issued by a core may contain errors in the target thread ID, the target memory address, and the data. In order to avoid that errors can modify the global state of the system, i.e. manipulating the synchronization count of a wrong thread or overwriting data at the wrong address, we propose to use a core-local ECC protected write buffer, similar to the write buffers proposed in [18]

for hardware transactional memory with pessimistic version management. The write buffer is managed by the L-TSU. The L-TSU buffers all TWRITEs issued by a core in its core-local write buffer. The write buffer keeps all TWRITEs until the L-TSU receives a message from the D-TSU to commit the buffered TWRITEs. The L-TSU then forwards all TWRITEs to the D-TSU, which processes them in the usual way, i.e. decreases the target threads' synchronization counts and stores the TWRITE data in the target thread frames. As mentioned, the L-TSU only permits the leading thread to issue persistent memory accesses (TWRITEs). In particular, the write buffer and the speculatively created continuations in the D-TSU ensure fault isolation between dataflow threads and prevent subsequent threads to consume wrong results. Furthermore, the L-FDU attached to each core creates the CRC-32 signatures of the redundant threads, incorporating the target thread IDs, the addresses and data of all TWRITEs, similar to the fingerprint technique proposed by Smolens et al. [41].

5.3 Comparison with Control-Flow-Based Redundant Execution Approaches

Redundant execution mechanisms for control-flow-based architectures have been widely studied with the advent of chip multiprocessors. However, control-flow based redundant execution in combination with parallel applications poses significant challenges for input replication, redundant thread synchronisation, and output comparison. In this section, we compare double execution of dataflow threads with Reunion [40] and DCC [25], two redundant execution approaches for chip multiprocessors.

5.3.1 Reunion

Reunion [40] is a loosely-coupled redundant execution scheme for shared-memory multicore architectures. The approach connects cores in a multicore architecture to redundant execution pairs, where the *vocal core* is allowed to issue stores and update the coherence system, while the *mute core* is only allowed to read from the memory, without updating or manipulating the global memory state. The detection of faults is restricted to the out-of-order pipeline and updates to the register file and the store buffer must be checked for faults, in order to re-use the roll-back capability of the out-of-order pipeline for recovery. Reunion uses *Relaxed Input Replication* for input replication, which allows redundant execution pairs to consum possible diverging data by concurrent memory accesses. Possible divergence caused by relaxed input replication is detected by comparing the computational results of the *vocal* and the *mute* core. However, unlike transient faults, diverging computational results due to concurrent memory accesses can not always be solved by re-execution. In this case, Reunion employs an expensive instruction-level lockstepping approach to guarantee forward progress.

5.3.2 Dynamic Core Coupling (DCC)

DCC [25] is another loosely-coupled redundant execution scheme, which supports dynamic core coupling. Fault detection is restricted to the cores' pipelines, where

computational results must be compared at each store. DCC uses cache-locking to buffer results in the private caches of the cores, which supports comparison intervals of more than 10,000 cycles.

To ensure consistent input replication among the redundant execution pairs, DCC extends the coherence protocol, which maintains a *master-slave memory access window* and an *age table* per core to guarentee consistent values in redundantly executed threads. In particluar, the *master-slave memory window* enforces an ordering of the memory accesses of the master and slave threads and detects possible violations for consistent input replication, ultimately delaying stores that may violate them.

5.3.3 Comparison with Double Execution

Reunion uses the microarchitecture of the cores to store possible values until comparison, which means that the synchronisation interval is restricted by the capacity of the pipeline to buffer computational results. On the other side, result comparison must have low latency, since the vocal core can only make forward progress, when its computations have been checked by the mute core. Furthermore, LaFrieda et al. [25] have shown that modern parallel workloads may induce a high performance penalty for *Relaxed Input Replication*, due to frequent inconsistencies between asynchronous threads. LaFrieda et al. [25] finally conclude that *Relaxed Input Replication* is the main source for degrading redundant execution performance in highly-parallel applications. Although LaFrieda et al. [25] have shown that their input replication based on an extended coherence protocol can outperform *Relaxed Input Replication*, Sanchez et al. [37] later showed hat DCC requires fast result comparison, which makes it use in tiled architectures, which communicate by a network-on-a-chip, inefficient and may induces high overhead for redundant execution. Although Relaxed Input Replication does not require complex hardware to duplicate data, it requires lockstepping in the case of dataraces between redundant stores, increasing communication among cores and complicating redundant thread management. Strict input replication provided by LaFrieda et al. [25] reduces the performance impact due to recovery of relaxed input replication, but complicates the hardware design by an enhanced coherence protocol. Furthermore, the solution can not be used in parallel architectures with long latencies [37].

If we compare both approaches with double execution, it can be seen that the dataflow semantics simplifies the effort to ensure consistent input replication for stores, since intervening stores are impossible and input replication must only be guarenteed for TSCHEDULE instructions. This also simplifies the management of the thread scheduling, since redundant threads can be executed independently from each other. Furthermore, result comparison and synchronization overhead is reduced, because only the *write set* of a thread must be compared. By contrast, Reunion needs to create the fingerprint signature over the register write backs and the stores in the store buffer, while DCC must create the signature over all stores to the first-level cache and, at the end of each comparison interval, of the architectural registers of the pipeline.

Double execution does not only simplify input replication, but also has some drawbacks, since synchronisation cannot be adopted at runtime, which means that the

synchronisation and comparison frequency are determined statically by the structure of the dataflow application.

6 Fault Recovery

For the fault recovery scheme we exploit the dataflow execution model as well, which provides inherent execution checkpoints between threads. Hence, threads can be restarted as long as their TWRITEs are not visible to the global system state and the effect of all previously issued TSCHEDULE instructions can be reverted. Since the side-effect free execution is supported by the coarse-grained dataflow execution model and fault isolation is guaranteed through the core-local write buffers and the speculatively created continuations, the D-TSU can restart dataflow threads to recover from faults.

The cost for the thread restart mechanism during fault free execution depends on the size of the written data. The overhead is mainly induced by the deferred TWRITEs introduced with the core-local write buffers. Therefore, a thread accesses the global frame memory only when a TDESTROY instruction was called and the local-write buffer is going to commit. This means, during thread execution, there are no TWRITE accesses to the global frame memory. We assume that writes to the core-local TWRITE buffer have low latency, similar to first level cache accesses. However, after TDESTROY has been reached and the D-FDU confirms the fault free execution, the entire local-write buffer must be committed to subsequent thread frames and the threads synchronization count is decreased. The overhead when a fault has been detected is mainly determined by the wasted execution time of the recovered thread. Since we only compare thread results when TDESTROY has been called, the wasted execution time in turn mainly depends on the length of the thread. This means, although the recovery mechanism is transparent to the programmer and the compiler, its recovery capability is constrained by the length of the dataflow threads.

Figure 7 shows how the recovery mechanism will work. Note that we implicitly assume double execution to detect faults. When the D-FDU determines a fault within

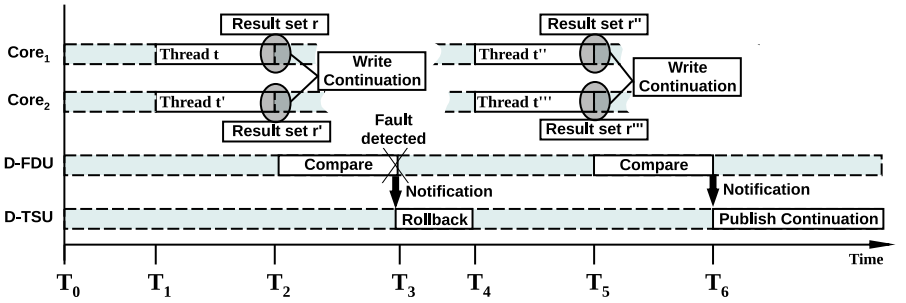

Fig. 7 Example for a recovery scenario. Thread t is duplicated as t and t'. After the execution both thread instances write back their result set, which are compared by the D-FDU (at T_2). In the case of a transient fault, the D-TSU re-executes Thread t as t'' and t''' again on the same cores (T_4)

a monitored core (between time T_2 and T_3), it provides the corresponding core ID, together with the fault information to its affiliated D-TSU. Subsequently, the D-FDU tries to determine the cause of the detected fault. Depending on the kind of the fault the D-TSU can either restart the thread (at T_4, after the rollback between time T_3 and T_4) on the same core or re-allocate all threads of the faulty core to reliable cores. In the given case of a transient fault, usually the D-TSU will try to re-execute a thread again on the original core (at T_4). The re-execution can easily be done by overwriting the continuation field at the L-TSU with the redundant continuation field hold by the D-TSU. The L-TSU will then schedule the thread again.

If the D-FDU assumes a permanent or intermittent fault due to many re-execution attempts or information from the L-FDU, it must exclude the faulty core from further workload. This is done by providing the D-TSU with the information about the identity of the faulty core. Consequently, the D-TSU re-schedules all threads of the faulty core on another reliable core. In order to do so, the D-TSU traverses its TCL and searches for corresponding continuations scheduled to the faulty core. If the D-TSU finds an entry that is associated with the faulty core, it reassigns the entry to a reliable core.

7 Evaluation

7.1 Experimental Setup

We evaluated double execution and thread recovery techniques with HP's COTSon multi-core simulator [5], which was extended with support for coarse-grained dataflow execution. Therefore, we implemented the D-TSU and the D-FDU to provide runtime support for double execution and thread restart recovery within the D-TSU. The goal of this evaluation was to investigate the overhead introduced by double execution in comparison with normal dataflow execution. Finally, we explored the overhead of the thread restart recovery without double execution, assuming a core-internal fault detection mechanism, under different core failure rates.

All experiments were limited to one node. This limitation is without loss in generality for the evaluation of the fault tolerance mechanisms, since redundant execution and recovery are restricted to one node. This means, in case of an optimal system wide load balancing between the nodes, the overhead of the complete system can be derived from the overhead per node. Furthermore, since redundant threads are always executed on the same node, only TSCHEDULE and TWRITEs contribute to the inter-node traffic. As a consequence, the inter-node traffic for double execution is nearly the same as for a regular dataflow execution, except for the additional TSCHEDULEs of the redundant threads. The baseline machine assembles a contemporary multi-core processor with 4, 8, 16, or 32 cores, respectively. Each core consists of an out-of-order pipeline with 5 stages and a maximal fetch and commit width of 2 instructions per cycle. The private cache hierarchy of each core is comprised of separate 32 kB L1 instruction and data caches and a 256 kB unified L2 cache. All cores have access to a 16 MB FM, exclusively used to store thread frames. The assumed memory bus latency is 25 cycles, while the memory latency is 100 cycles. Table 1 depicts the parameters of the baseline machine in more detail.

Table 1 Hardware parameters of the simulated machine

Parameters	Values
Cores	4, 8, 16, 32
Core parameters	Out-of-order, pipeline length: 5, Fetch width: 2, Commit width: 2
L1 I- and D-cache (private per core)	Size: 32 kB, line size: 64, hit latency: 6 cycles
Unified L2-cache (private per core)	Size: 256 kb, line size: 64, sets: 8, hit latency: 10 cycles
Memory bus latency (L3-cache to memory	25 cycles
Memory latency	100 cycles
TWRITE latency (write buffer)	3 cycles
TWRITE latency (commit to memory)	30 cycles
TSCHEDULE latency	40 cycles
TDESTROY latency	40 cycles
D-FDU Signature comparison latency	30 cycles

We further assume that writing to the core-private write buffer and generating a CRC-32 signature takes 3 cycles per TWRITE. Committing one TWRITE instruction in the write buffer to global memory is supposed to take 30 cycles. This is assumed to be faster than a regular memory access, since committing the write buffer may take advantage of the DRAMs burst mode. Finally, for the TSCHEDULE instruction, we assume a latency of 40 cycles. The simulated node was stressed with two coarse-grained dataflow benchmarks: a parallel version of Fibonacci (fib), which recursively calculates the Fibonacci numbers, and a block-wise matrix multiply algorithm (matmul). fib is a modified version of recursive Fibonacci, which spawns three new threads in each recursive step. Nonetheless, fib recursively spawns new threads for $(n - 1)$ and $(n - 2)$ until n is equal or less than 28. If n is equal or less than 28, Fibonacci of n is calculated locally. matmul is a standard block-wise matrix multiplication, in which the maximum thread-level parallelism is restricted by the number of blocks per matrix, while the length of each dataflow thread is determined by the block size. Both benchmarks fully support the T*-instruction set extension [14] and therefore guarantee a side-effect free execution. This enables the simulator to restart and recover from all faults detected.

7.2 Double Execution Overhead

We simulated fib(36) and fib(40) with the goal to measure the impact of core utilization on the runtime overhead introduced by double execution. Here, fib(36) serves as a benchmark, which cannot completely utilize the nodes cores. To show the increasing execution time with double execution, we determined the performance degradation, which is the difference of double execution time (*TDX*) and the regular dataflow execution time (*TDF*) normalized to the regular dataflow execution time on the same node, i.e.

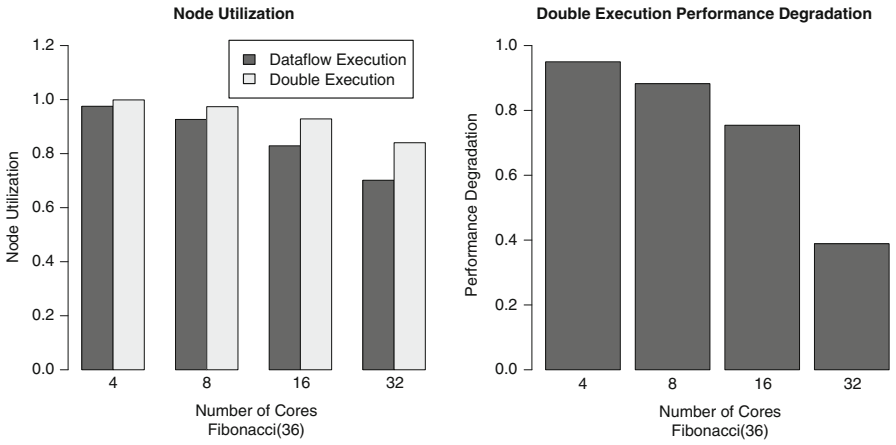

Fig. 8 Node utilization (*left*) and performance degradation for double execution of Fibonacci(36)

$$Performance\ Degradation = \frac{(TDX - TDF)}{TDF}.$$

The plot on the left side of Fig. 8 shows the core utilization of regular dataflow execution and of double execution. It can be seen that fib(36) is able to nearly utilize 4 cores. However, with an increasing node size, fib(36) is no longer able to utilize all cores. The plot on the right side of Fig. 8 depicts the performance degradation for the different node configurations. We can see that double execution can exploit underutilized cores to speed up double execution runtime. Performance degradation for 4 cores is high, because all 4 cores are already nearly fully utilized by fib(36) and double execution can not make use of underutilized cores. For 32 cores, the runtime overhead introduced by double execution of fib(36) is only 38.8 %, because underutilized cores can be used for spatial redundancy.

The plot of fib(40) (see Fig. 9) in turn shows that even if double execution cannot use underutilized cores (all configurations are utilized by 99 %), the performance degradation never exceeds 100 %. In a tightly-coupled lockstep system, one would expect an overhead of above 100 %, due to the doubled workload and the overhead for result comparison. However, the scalability of the execution model can efficiently handle the increased workload and hides the overhead introduced with result comparison and defered result propagation.

Matrix multiply of two 256 × 256 matrices (shown in Fig. 10) revealed that double execution may also benefit from the locality in the nodes caches. Even though the benchmark is able to fully utilize all node configurations from 4 to 32 cores in dataflow execution, the performance degradation of double execution decreases with an increasing number of cores per node.

7.3 Thread Restart Recovery

We simulated all benchmarks with a Mean Time to Failure (MTTF) of 0.1 and 0.01 s per core. Since we assume a constant failure rate per core, MTTFs of 0.1 and 0.01 s result in

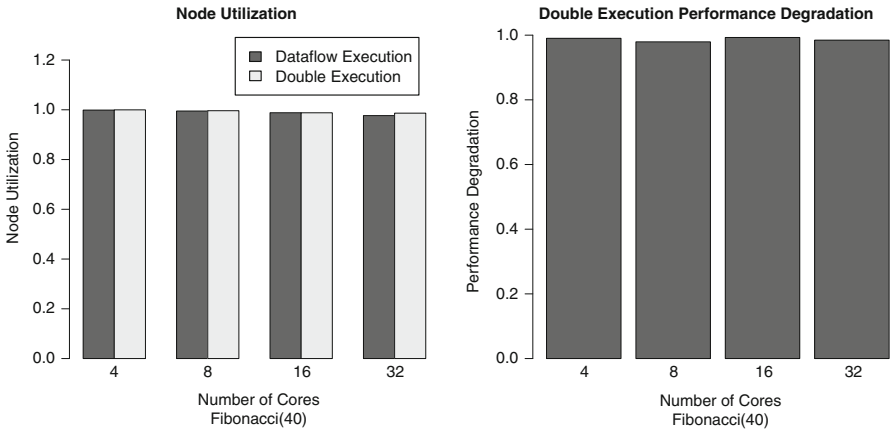

Fig. 9 Node utilization (*left*) and performance degradation for double execution of Fibonacci(40)

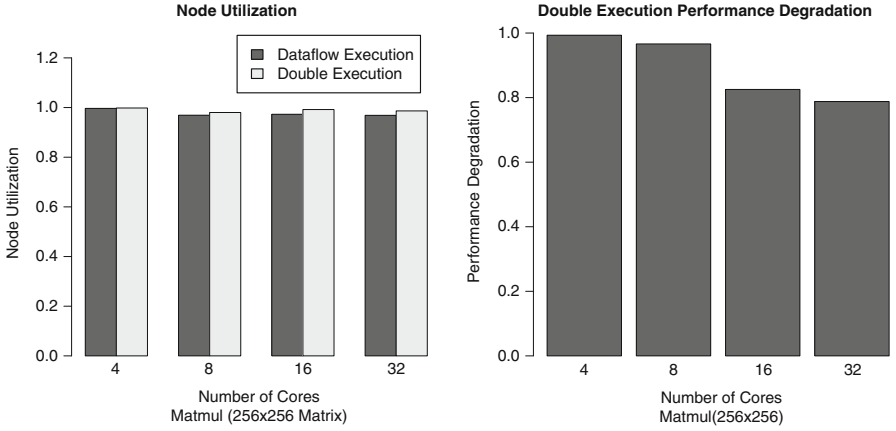

Fig. 10 Node utilization (*left*) and performance degradation for double execution of Matmul (256×256 matrices)

average failure rates of 10 failures and 100 failures per second per core. Please note that we do not use double execution here in order to measure the pure overhead introduced with thread restart recovery. We normalized the execution times to a failure free run on the same node. For an MTTF of 0.1 s (see Fig. 11, left) it can be seen that Fibonacci suffers from a performance degradation of 10–20 %. Furthermore, the performance degradation is nearly constant with increasing node sizes. The overall overhead for both Fibonacci benchmarks was 17.9 %. The performance degradation of matmul, however, decreases until a node size of 16. However, the performance degradation for 4 cores is considerably higher than for Fibonacci. The average overhead of all node configurations for the matmul benchmark was 27.9 %. By reducing the MTTF to 0.01 s (see Fig. 11, right), we can see a performance degradation between 150 and 220 % for

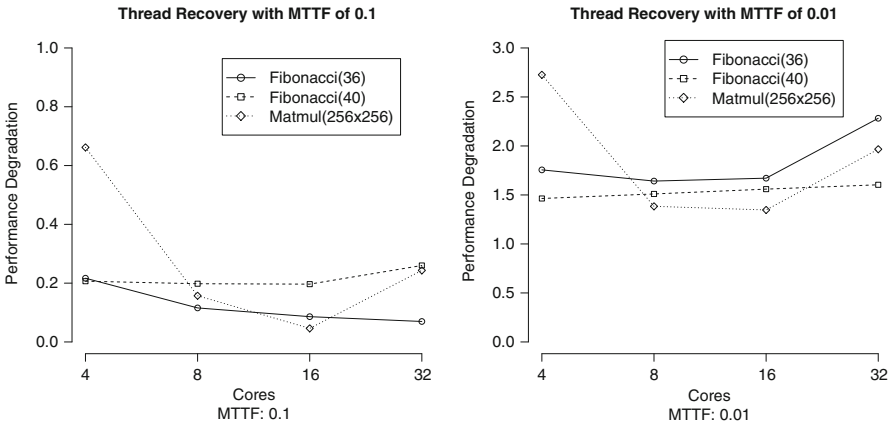

Fig. 11 Performance degradation due to thread recovery with MTTFs of 0.1 s (*left*) and 0.01 s (*right*)

Fibonacci. The average overhead for Fibonacci was 187.4 %. The average overhead for matmul was 185.5 %.

8 Conclusion

This paper presented a fault-tolerant coarse-grained dataflow architecture, which is able to dynamically cope with transient, intermittent, and permanent faults on all levels of a future scalable many-core system.

Runtime fault management is provided by the intra-node D-FDUs, which monitor cores and other D-FDUs in order to dynamically react and adapt the system in the case of faults. Fault Detection is implemented by double execution, which detects faults in the cores' pipeline and by exploiting the machine check architecture, which detects faults in on-chip memories, buses, and interconnects. Furthermore, we leveraged the dataflow execution model, in particular its side-effect free and single-assignment semantics, for a core-local recovery mechanism using thread restarts.

Experimental results showed that the coarse-grained dataflow system can efficiently make use of underutilized cores to speed up double execution. In the case of a fully utilized system, the double execution overhead is still smaller than in a comparable lockstep architecture.

In a system suffering from an MTTF of 0.1 s, thread-restart recovery introduced an average overhead of 17.9 % for Fibonacci and 27.9 % for matrix multiplication, while for an MTTF of 0.01 s the overhead was 197.4 and 185.5 %, respectively. Since already a MTTF of 0.1 s is an artificial high failure rate, it can be seen that thread-restart recovery can be efficiently used in coarse-grained dataflow systems, even with increasing failure rates.

Acknowledgments This work was partly funded by the European FP7 Projects TERAFLUX (id. 249013) and HiPEAC (IST-217068). The authors wish to thank N. Puzovic and Z. Popovic for their initial studies on the DTA-C architecture and P. Faraboschi of HP for his precious suggestions and support on the COTSon simulator.

References

1. International Technology Roadmap for Semiconductors 2011 Edition. Website. http://www.itrs.net
2. Agarwal, R., Garg, P., Torrellas, J.: Rebound: scalable checkpointing for coherent shared memory. In: International Symposium on Computer Architecture (ISCA), pp. 153–164. IEEE (2011)
3. AMD Inc.: AMD64 Architecture Programmer's Manual Volume 2: System Programming (2006)
4. Arandi, S., Kyriacou, C., George, M., George, M., Masrujeh, N., Trancoso, P., Evripidou, S., Giorgi, R., Zhibin, Y., Collange, S., Scionti, A., Khan, B., Khan, S., Lujan, M., Watson, I., Etsion, Y., Ungerer, T., Fechner, B., Garbade, A., Weis, S.: D6.2-advanced teraflux architecture. Public deliverable, The TERAFLUX Project (FP7/2007-2013 Grant Agreement No. 249013) (2011)
5. Argollo, E., Falcón, A., Faraboschi, P., Monchiero, M., Ortega, D.: COTSon: infrastructure for full system simulation. ACM SIGOPS Oper. Syst. Rev. **43**(1), 52–61 (2009)
6. Austin, T.: DIVA: a reliable substrate for deep submicron microarchitecture design. In: International Symposium on Microarchitecture (MICRO), pp. 196–207. IEEE (1999)
7. Bell, S., et al.: TILE64-processor: a 64-core soc with mesh interconnect. In: International Solid-State Circuits Conference (ISSCC). Digest of Technical Papers, pp. 88–89. IEEE (2008)
8. Bernick, D., Bruckert, B., Vigna, P., Garcia, D., Jardine, R., Klecka, J., Smullen, J.: Nonstop advanced architecture. In: International Conference on Dependable Systems and Networks (DSN), pp. 12–21. IEEE (2005)
9. Borkar, S.: Designing reliable systems from unreliable components: the challenges of transistor variability and degradation. IEEE Micro **25**(6), 10–16 (2005)
10. Borkar, S.: Thousand core chips: a technology perspective. In: Annual Design Automation Conference (DAC), pp. 746–749. ACM (2007)
11. Elnozahy, E.N.M., Alvisi, L., Wang, Y.M., Johnson, D.B.: A survey of rollback-recovery protocols in message-passing systems. ACM Comput. Surv. **34**(3), 375–408 (2002)
12. Etsion, Y., Cabarcas, F., Rico, A., Ramirez, A., Badia, R. M., Ayguade, E., Labarta, J., Valero, M.: Task superscalar: an out-of-order task pipeline. In: International Symposium on Microarchitecture (MICRO), pp. 89–100. IEEE (2010)
13. Gautier, T., Besseron, X., Pigeon, L.: KAAPI: a thread scheduling runtime system for data flow computations on cluster of multi-processors. In: International Workshop on Parallel Symbolic Computation (PASCO), pp. 15–23. ACM (2007)
14. Giorgi, R.: TERAFLUX: exploiting dataflow parallelism in teradevices. In: International Conference on Computing Frontiers (CF), pp. 303–304. ACM (2012)
15. Giorgi, R., Popovic, Z., Puzovic, N.: DTA-C: a decoupled multi-threaded architecture for CMP systems. In: International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD), pp. 263–270. IEEE (2007)
16. Giorgi, R., Popovic, Z., Puzovic, N.: Implementing fine/medium grained TLP support in a many-core architecture. In: Bertels, K., Dimopoulos, N., Silvano, C., Wong, S. (eds.) Embedded Computer Systems: Architectures, Modeling, and Simulation, Lecture Notes in Computer Science (LNCS), vol. 5657, pp. 78–87. Springer (2009)
17. Gupta, G., Sohi, G.S.: Dataflow execution of sequential imperative programs on multicore architectures. In: International Symposium on Microarchitecture (MICRO), pp. 59–70. ACM (2011)
18. Hammond, L., Wong, V., Chen, M., Carlstrom, B.D., Davis, J.D., Hertzberg, B., Prabhu, M.K., Wijaya, H., Kozyrakis, C., Olukotun, K.: Transactional memory coherence and consistency. In: International Symposium on Computer Architecture (ISCA), pp. 102–113. IEEE (2004)
19. Howard, J., et al.: A 48-core ia-32 message-passing processor with dvfs in 45nm CMOS. In: International Solid-State Circuits Conference (ISSCC). Digest of Technical Papers, pp. 108–109. IEEE (2010)
20. Hum, H.H.J., Maquelin, O., Theobald, K.B., Tian, X., Tang, X., Gao, G.R., Cupryk, P., Elmasri, N., Hendren, L.J., Jimenez, A., Krishnan, S., Marquez, A., Merali, S., Nemawarkar, S.S., Panangaden, P., Xue, X., Zhu, Y.: A design study of the EARTH multiprocessor. In: International Conference on Parallel Architectures and Compilation Techniques (PACT), pp. 59–68. IFIP Working Group (1995)
21. Iyer, R., Nakka, N., Kalbarczyk, Z., Mitra, S.: Recent advances and new avenues in hardware-level reliability support. IEEE Micro **25**(6), 18–29 (2005)
22. Jafar, S., Gautier, T., Krings, A., louis Roch, J.: A checkpoint/recovery model for heterogeneous dataflow computations using work-stealing. In: Cunha, J.C., Medeiros P.D.(eds.) Euro-Par 2005 Parallel

Processing, Lecture Notes in Computer Science (LNCS), vol. 3648, pp. 675–684. Springer, Berlin, Heidelberg (2005)
23. Kelm, J.H., Johnson, D.R., Johnson, M.R., Crago, N.C., Tuohy, W., Mahesri, A., Lumetta, S.S., Frank, M.I., Patel, S.J.: Rigel: an architecture and scalable programming interface for a 1000-core accelerator. In: International Symposium on Computer Architecture (ISCA), pp. 140–151. IEEE (2009)
24. Kephart, J., Chess, D.: The vision of autonomic computing. Computer **36**(1), 41–50 (2003)
25. LaFrieda, C., Ipek, E., Martinez, J., Manohar, R.: Utilizing dynamically coupled cores to form a resilient chip multiprocessor. In: International Conference on Dependable Systems and Networks (DSN), pp. 317–326. IEEE (2007)
26. Lee, B., Hurson, A.R.: Dataflow architectures and multithreading. Computer **27**(8), 27–39 (1994)
27. Li, F., Pop, A., Cohen, A.: Automatic extraction of coarse-grained data-flow threads from imperative programs. IEEE Micro **32**(4), 19–31 (2012)
28. Mukherjee, S.S., Kontz, M., Reinhardt, S.K.: Detailed design and evaluation of redundant multithreading alternatives In: International Symposium on Computer Architecture (ISCA), pp. 99–110. IEEE (2002)
29. Nguyen-tuong, A., Grimshaw, A.S., Hyett, M.: Exploiting data-flow for fault-tolerance in a wide-area parallel system. In: International Symposium on Reliable and Distributed Systems, pp. 1–11 (1996)
30. Prvulovic, M., Zhang, Z., Torrellas, J.: Revive: cost-effective architectural support for rollback recovery in shared-memory multiprocessors. In: International Symposium on Computer Architecture (ISCA), pp. 111–122. IEEE (2002)
31. Rashid, M., Huang, M.: Supporting highly-decoupled thread-level redundancy for parallel programs. In: International Symposium on High Performance Computer Architecture (HPCA), pp. 393–404. IEEE (2008)
32. Ray, J., Hoe, J.C., Falsafi, B.: Dual use of superscalar datapath for transient-fault detection and recovery. In: International Symposium on Microarchitecture (MICRO), pp. 214–224. IEEE (2001)
33. Reinhardt, S.K., Mukherjee, S.S.: Transient fault detection via simultaneous multithreading. In: International Symposium on Computer Architecture (ISCA), pp. 25–36. ACM (2000)
34. Rotenberg, E.: AR-SMT: a microarchitectural approach to fault tolerance in microprocessors. In: Twenty-Ninth Annual International Symposium on Fault-Tolerant Computing, 1999. Digest of Papers, pp. 84-91 (1999)
35. Sánchez, D., Aragón, J., García, J.: Evaluating dynamic core coupling in a scalable tiled-cmp architecture. In: International Workshop on Duplicating, Deconstructing, and Debunking (WDDD) (2008)
36. Sánchez, D., Aragón, J., García, J.: Extending SRT for parallel applications in tiled-CMP architectures. In: International Symposium on Parallel and Distributed Processing (IPDPS), pp. 1–8. IEEE (2009)
37. Sánchez, D., Aragón, J.L., García, J.M.: REPAS: Reliable Execution for Parallel Applications in Tiled-CMPs. In: Sips, H., Epema, D., Lin, H.X. (eds.) International Euro-Par Conference on Parallel Processing, Lecture Notes in Computer Science (LNCS), vol. 5704, pp. 321–333. Springer, Berlin, Heidelberg (2009)
38. Sánchez, D., Aragón, J. L., García, J.M.: A log-based redundant architecture for reliable parallel computation. In: International Conference on High Performance Computing (HiPC), pp. 1–10. IEEE (2010)
39. Slegel, T., Averill, R.M.I., Check, M., Giamei, B., Krumm, B., Krygowski, C., Li, W., Liptay, J., Macdougall, J., McPherson, T., Navarro, J., Schwarz, E., Shum, K., Webb, C.: IBM's S/390 G5 microprocessor design. IEEE Micro **19**(2), 12–23 (1999)
40. Smolens, J.C., Gold, B.T., Falsafi, B., Hoe, J.C.: Reunion: complexity-effective multicore redundancy. In: International Symposium on Microarchitecture (MICRO), pp. 223–234. IEEE (2006)
41. Smolens, J.C., Gold, B.T., Kim, J., Falsafi, B., Hoe, J.C., Nowatzyk, A.G.: Fingerprinting: bounding soft-error detection latency and bandwidth. In: International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), pp. 224–234. IEEE (2004)
42. Sorin, D.J., Martin, M.M.K., Hill, M.D., Wood, D.A.: Safetynet: improving the availability of shared memory multiprocessors with global checkpoint/recovery. In: International Symposium on Computer Architecture (ISCA), pp. 123–134. IEEE (2002)
43. Srinivasan, J., Adve, S.V., Bose, P., Rivers, J.A.: The impact of technology scaling on lifetime reliability. In: International Conference on Dependable Systems and Networks (DSN), pp. 177–186. IEEE (2004)
44. Stavrou, K., Evripidou, P., Trancoso, P.: DDM-CMP: Data-Driven Multithreading on a Chip Multiprocessor. In: Hämäläinen, T.D., Pimentel, A.D., Takala, J., Vassiliadis, S. (eds.) Embedded Computer

Systems: Architectures, Modeling, and Simulation, Lecture Notes in Computer Science (LNCS), vol. 3553, pp. 364–373. Springer, Berlin, Heidelberg (2005)
45. Weis, S., Garbade, A., Schlingmann, S., Ungerer, T.: Towards fault detection units as an autonomous fault detection approach for future many-cores. In: ARCS 2011 Workshop Proceedings, pp. 20–23. VDE (2011)
46. Weis, S., Garbade, A., Wolf, J., Fechner, B., Mendelson, A., Giorgi, R., Ungerer, T.: A fault detection and recovery architecture for a teradevice dataflow system. In: International Workshop on Data-Flow Execution Models for Extreme Scale Computing (DFM), pp. 38–44. IEEE (2011)
47. Wittenbrink, C., Kilgariff, E., Prabhu, A.: Fermi GF100 GPU architecture. IEEE Micro **31**(2), 50–59 (2011)
48. Yeh, Y.: Triple-triple redundant 777 primary flight computer. In: Proceedings of the Aerospace Applications Conference, pp. 293–307. IEEE (1996)
49. Zuckerman, S., Suetterlein, J., Knauerhase, R., Gao, G.R.: Using a "codelet" program execution model for exascale machines: position paper. In: Proceedings of the International Workshop on Adaptive Self-Tuning Computing Systems for the Exaflop Era (EXADAPT), pp. 64–69. ACM (2011)